Developing and Mechanizing Semantic Models for Program Logics

Aquinas Hobor[1] Robert Dockins[2]

[1]National University of Singapore

[2]Princeton University

Nov. 28, 2010 / APLAS 2010

Outline

Outline

Major Goals for this Tutorial

- Review two program logics:
  1. Types in the polymorphic λ-calculus
  2. Separation logic
- Discuss a program logic that combines features from both:
  3. concurrent separation logic with first-class locks
- As we go we show how to develop semantic models for these logics. We demonstrate the first two as self-contained examples; we give the third as an outline of a more complicated system.
- We are particularly interested in mechanization in Coq

Why use semantic methods?

- Two approaches to proving soundness: Semantic and Syntactic
- In the last 20 years, syntactic methods (Wright-Felleisen, *e.g.*, induction on typing judgments) have been more popular
- We prefer semantic methods—developing models of our program logics and proving the logics sound w.r.t. the model
- 10 years ago, syntactic methods had major advantages for people who wanted desirable program logic features like impredicative quantification. However, the state of the art has advanced—good semantic solutions are now available and ready for deployment.

Why use semantic methods?

- There is some evidence that semantic methods work better for larger projects, *e.g.* with better modularity and scaling [BDH+08].
- Semantic methods can be easier in mechanized settings; *e.g.*, most mechanized Hoare logics have semantic-style assertions, even when the soundness proofs are done syntactically.
- For high-reliability applications, the trusted computing base can be smaller since semantic methods require theorem checkers whereas syntactic methods require *meta*theorem checkers
- Still, at the end, it is still a matter of taste—if you have not tried semantic methods then why not give them a try!

Why use **our** semantic methods?

- We (and others at Princeton) have been focused on developing and mechanizing semantic methods for 10 years.
- Our techniques are available as a (BSD-licensed) library, so they can be rapidly applied to a new project. We have released several examples that can be modified for this purpose.
- Good “proof engineering”—techniques that help mechanization.
- It is relatively easy to develop toy languages and models that completely break when you want to make them more realistic. The techniques we have developed will scale to fully-realistic settings.
- We continue to improve the library; you will be able to take advantage without having to mechanize those techniques yourself.

Approximation and Separation

Approximation is a term we use to mean a certain collection of mathematical tools which solve knotty problems arising when one needs to associate invariants with memory locations. The classic example is λ calculus with references; the invariants are the types of the reference cells.

Separation refers to a fairly recent idea of explicitly reasoning about disjoint or nonoverlapping resources in a program logic. Its primary use is to cleanly handle the issue of pointer nonaliasing in languages with addressable memory. It is especially useful for describing and reasoning about inductive (tree-structured) data structures.

Citations for Approximation

- Semantics of Types for Mutable State. Amal J. Ahmed. Princeton University PhD thesis TR-713-04, 2004.
- A Very Modal Model of a Modern, Major, General Type System. Andrew W. Appel, Paul-Andre Melliès, Christopher D. Richards, and Jerôme Vouillon. POPL 2007, January 2007, 109–122.
- A Theory of Indirection via Approximation. Aquinas Hobor, Robert Dockins, and Andrew W. Appel. POPL 2010, Jan 2010, 171–185.

A promising alternate approach based on metric spaces (will not be discussed in this talk):

- The category-theoretic solution of recursive metric-space equations. L. Birkedal, K. Støvring, and J. Thamsborg. Theoretical Computer Science, 411:4102-4122, 2010.
- Step-indexed kripke models over recursive worlds. 2010. L. Birkedal, B. Reus, J. Schwinghammer, K. Støvring, J. Thamsborg, and H. Yang. To appear, POPL 2011.

Citations for Separation

- Separation Logic: A Logic for Shared Mutable Data Structures. John Reynolds. LICS 2002, July 2002, 55–74.
- Resources, Concurrency and Local Reasoning. Peter W. O'Hearn. TCS vol. 375, May 2007, 271–307.
- Permission Accounting in Separation Logic. Richard Bornat, Cristiano Calcagno, Peter W. O'Hearn, and Matthew Parkinson. POPL 2005, January 2005, 259–270.
- Local Action and Abstract Separation Logic. C. Calcagno, P. W. O'Hearn, and H. Yang. LICS 2007, July 2007, 366-378.
- Oracle Semantics for Concurrent Separation Logic. Aquinas Hobor, Andrew W. Appel, and Francesco Zappa Nardelli. ESOP 2008, April 2008, 353–367.
- A Fresh Look at Separation Algebras and Share Accounting. Robert Dockins, Aquinas Hobor, and Andrew W. Appel. APLAS 2009, December 2009, 161–177.

Mechanized Semantic Library

- Coq library focused on developing semantic models for interesting program logics
- Current version: 0.3; we hope to get 0.4 out in the next few months
- Available at: `http://msl.cs.princeton.edu/`

MSL use cases

- Types in the polymorphic λ-calculus with references
- Separation logic for imperative while programs
- Concurrent C minor / Verified Software Toolchain
  Extending C minor to concurrency; connecting machine-verified source program verifications to certified targets.
- Barriers in Concurrent Separation Logic
  An extension of CSL to Pthreads-style barriers.
- A Theory of Termination via Indirection
  Total correctness with function pointers.
- Heap-Hop
  Separation logic verification tool, uses our extracted share model

Relationship of paper to tutorial

- In addition to this tutorial, we have prepared a paper that is included in the APLAS proceedings entitled *A Logical Mix of Approximation and Separation* (by Hobor, Dockins, and Appel)
- This tutorial is "example heavy"—the paper is "theory heavy"
- Hopefully in this tutorial we can give you a good feel for how to develop our style of semantic models, and then you can use the paper as a reference for the more technical aspects
- Both tutorial slides and paper are available on MSL website

Outline

Example 1: Main take-away ideas

- Trusted Computing Base
- Semantics of approximation to model indirection
- End-to-end soundness proofs

The polymorphic λ-calculus

You are probably familiar with the polymorphic λ-calculus (System F). Here is the flavor we will be using today:

expressions	e	$=$	n	natural numbers
		$\mid$	$f(e)$	primitives; $f : \mathbb{N} \to e$
		$\mid$	x	variables; de Bruijn encoding in Coq
		$\mid$	ℓ	addresses
		$\mid$	$\lambda x.e$	functions; just $\lambda.e$ in Coq
		$\mid$	$e_1\ e_2$	application
		$\mid$	$\mathsf{new}(e)$	allocation
		$\mid$	$!e$	dereference
		$\mid$	$e_1 := e_2; e_3$	update

We have added a few features (primitives, building the sequence into the update operation, etc.) to make writing example programs simpler.

One key omission may not be familiar. What is missing?

Curry vs. Church

The polymorphic λ-calculus is often presented with explicit type abstraction and application, often using Λ and $e[\tau]$ respectively.

This is called a *Church-style* calculus: type operations appear in the syntax and operational steps are required to evaluate type application.

$$(\Lambda\tau.\lambda x : \tau.x)\ [\mathsf{int}]\ 3 \quad \mapsto \quad (\lambda x : \mathsf{int}.x)\ 3 \quad \mapsto \quad 3$$

In contrast, we have a *Curry-style* calculus: program syntax does not contain types, and applying a polymorphic function to an argument does not require an extra operational step.

$$(\lambda x.x)\ 3 \quad \mapsto \quad 3$$

Curry vs. Church (2)

Why is this important?

Because real machines (*e.g.*, x86) do not take operational steps to evaluate types. We want our techniques to scale to the point where we can typecheck machine code. If we are only able to handle Church-style calculi, then we will have to insert dummy instructions (nop) into our machine code so that we can properly typecheck.

This leads to inferior performance—and is ugly. We need to be confident that our technique can handle Curry-style calculi.

Basic semantics for the λ-calculus

We require several basic semantic ideas, largely unsurprising:

- Closed expressions: no free variables
- Values: naturals n, addresses ℓ, and **closed** λ-terms $\lambda x.e$ (in Coq, use dependent types)
- Memories m: a pair (b, ζ) of a *break* b (address used to track the boundary of unallocated memory) and a function ζ from addresses to values. Memories support the following operations:
  - $\mathsf{deref}(m, \ell)$: return the value $\zeta(\ell)$
  - $\mathsf{update}(m, \ell, v)$: return the memory $(b, [\ell \mapsto v]\zeta)$
  - $\mathsf{new}(m, v)$: return the pair $(b,\ (b+1, [b \mapsto v]\zeta))$
- States σ: pairs (m, e) of memory and expression

Basic semantics for the λ-calculus (2)

- A step relation $\sigma \mapsto \sigma'$:
  - Small-step call-by-value untyped λ-calculus
  - Using the memory operations deref, update, and new to evaluate $!e$, $e_1 := e_2$; e_3, and $\mathsf{new}(e)$, respectively
- Reflexive, transitive closure of same: $\mapsto^*$
- A *safety policy*:
  - A state σ *can step* if there exists σ' such that $\sigma \mapsto \sigma'$
  - A state σ is *safe* if for all reachable σ'—that is, $\sigma \mapsto^* \sigma'$—σ' has reached a value or can step
  - An expression e is a *safe program* if, for any memory m, the state (m, e) is safe
- **The goal of our type system will be to show that certain expressions (*i.e.*, those that are well-typed) are safe programs**

Trusted Computing Base

Together, these semantic definitions make up the *Trusted Computing Base* (TCB)—the definitions that need to be carefully examined to make sure that we are proving the theorem that we intend.

Errors in the TCB (for example, a mistake in the operational semantics of substitution; or an incorrect safety policy) will not be caught by a proof checker, since the TCB are the *assumptions* used in the proof.

Especially when doing a complex mechanized proof, it is important to get the TCB as small and simple as possible. In the case of our polymorphic λ-calculus with references, the TCB is 182 lines of Coq (including whitespace and comments—126 without).

Types

What is a type and what kinds of types do we want to support?

- A type, in the most general sense, is a way of classifying expressions according to that kinds of values they produce.
- We have only three kinds of values:
  - naturals n, for which we will want the type int
  - addresses ℓ, for which we will want the type $\mathsf{ref}\ \tau$
  - λ-terms, for which we will want the function type $\tau_1 \to \tau_2$
- In addition to those basic types, there are lots of other kinds of types we may want as well:
  - Quantified: universal $\forall$ and existential $\exists$
  - Logical: intersection $\cap$, union $\cup$, top $\top$, bottom $\bot$
  - Recursive: μF, for typing recursive structures such as lists
  - Other goodies: Singleton: (*e.g.*, $3 : \{3\}$); Subset (*e.g.*, for bounded quantification); Offset (for constructing records); ...

Impredicativity and Equirecursion

What is one allowed to quantify over? If you want a type system powerful enough to check, *e.g.*, the code emitted by an industrial-strength ML compiler, then one needs the more powerful kind of quantification, known as *impredicative*.

That is, we want to write $e : \forall\alpha.\ \tau$ where α can have any **metatype**—including, critically, that α is allowed to range over all of the types in our type system. (That is, in Coq, if α has some **metatype** A, then A : Type.) We need this feature to type check closures, which are a quantified package of environment and code; critically, the environment itself can contain closures (including, in the case of recursion through the store, for this function itself).

We also want *equirecursion*: full equality between μF and $F(\mu F)$. The alternative, *isorecursion*, is to require operational steps to roll and unroll the recursive types. If we want to check machine code, this would require extra nop instructions.

Types as sets; the trouble with ref

- For a simple λ-calculus, the types are just sets of values; for example, $\mathsf{int} \equiv \{0, 1, -1, 2, -2, \ldots\}$. Then $v : \tau$ just means $v \in \tau$.
- Sets in mechanized provers are encoded as functions:

$$\mathsf{int} \equiv \lambda v.\ \begin{cases} \top & \text{when } v = n \\ \bot & \text{when } v = \ell \\ \bot & \text{when } v = \lambda x.\ e \end{cases}$$

  Then $v : \tau$ really means $\tau(v)$ holds (in the metalogic)
- However, how can we define the type $\mathsf{ref}\ \tau$? We cannot say

$$\mathsf{ref}\ \tau \equiv \lambda v.\ \begin{cases} \top & \text{when } v = \ell \\ \bot & \text{otherwise} \end{cases}$$

  since then $\mathsf{ref\ int}$ is equal to $\mathsf{ref}\ (\mathsf{ref\ int})$

Memory typings

- Idea: we add a *memory typing*—a function ϕ from addresses to types. Our typing judgment is now $\phi \vdash v : \tau$, with types as sets of *worlds*—in this case, (memory typing,value) pairs:

$$\phi \vdash v : \tau \quad \equiv \quad (\phi, v) \in \tau \qquad \text{a.k.a.} \qquad \tau(\phi, v)$$

- Now we can try to define $\mathsf{ref}\ \tau$ as follows:

$$\mathsf{ref}\ \tau \equiv \lambda(\phi, v).\ \begin{cases} \phi(\ell) = \tau & \text{when } v = \ell \\ \bot & \text{otherwise} \end{cases}$$

- This seems like a good idea, but we've created a terrible contravariant cycle in our metatypes:

$$\begin{array}{rcl} \mathsf{type} & \equiv & (\mathsf{memtype} \times \mathsf{value}) \to \mathbb{T} \\ \mathsf{memtype} & \equiv & \mathsf{address} \rightharpoonup \mathsf{type} \end{array}$$

  A standard cardinality argument shows that no solution to this recursive definition exists in set theory.

Approximating the recursive equation

Although there is no exact solution, one might wonder about the existence of some "approximate" solution. Ten years ago, Ahmed developed the first step-indexed model based on this idea. It took 20,000 lines of HOL to mechanize.

Time marches on. The naïve attempt falls into the following pattern:

$$\begin{array}{rcl} F(X) & \equiv & \mathsf{address} \rightharpoonup X \\ O & \equiv & \mathsf{value} \\ \mathsf{memtype} & \approx & F((\mathsf{memtype} \times O) \to \mathbb{T}) \end{array}$$

We developed *indirection theory* to approximate any recursive equation that falls into this pattern (assuming F is covariant). Now it is possible to build the same (actually, a better) model that Ahmed developed in around 30 lines of Coq code. Just instantiate F and O in a module (here TFP) meeting a simple interface, and use:

```
Module K := KnotProp(TFP).
```

The knot

Indirection theory then builds a metatype knot (in Coq: `K.knot`) and an associated series of operations and definitions:

- knots are approximatable—that is, there is an "approximate" relation between worlds, written $w \rightsquigarrow w'$, and a "level" function from worlds to $\mathbb{N}$, written $|w|$, such that:
  - level of bottom: $(\nexists w'.\ w \rightsquigarrow w') \Rightarrow |w| = 0$
  - level of approximation: $w \rightsquigarrow w' \Rightarrow |w| = |w'| + 1$
  - weak unapproximation: $(\exists w.\ |w| = |w'| + 1) \Rightarrow \exists w.\ w \rightsquigarrow w'$
- a metatype $\mathsf{predicate} \equiv (\mathsf{knot} \times \mathsf{value}) \to \mathbb{T}$; predicate will be the metatype of types in our λ-calculus
- a *section-retraction* pair between the knot and the type $\mathbb{N} \times (\mathsf{address} \rightharpoonup \mathsf{predicate})$—that is, a pair of functions:
  - $\mathsf{squash} : (\mathbb{N} \times (\mathsf{address} \rightharpoonup \mathsf{predicate})) \to \mathsf{knot}$
  - $\mathsf{unsquash} : \mathsf{knot} \to (\mathbb{N} \times (\mathsf{address} \rightharpoonup \mathsf{predicate}))$
  - such that $\mathsf{squash} \circ \mathsf{unsquash}$ is the identity function, and
  - $\mathsf{unsquash} \circ \mathsf{squash}$ is a kind of approximation function

Approximating a predicate

What kind of approximation are we talking about? The key is an approximation defined on predicates as follows:

$$\mathsf{approx}_n(P) : \mathsf{predicate} \equiv \lambda k. \begin{cases} P(k) & \text{when } |k| < n \\ \bot & \text{otherwise} \end{cases}$$

The idea is that approx_n "forgets" how P behaves on knots of level $\geq n$:

$$\mathsf{approx}_2\left(\begin{cases} P_0 & \text{when } |k| = 0 \\ P_1 & \text{when } |k| = 1 \\ P_2 & \text{when } |k| = 2 \\ P_3 & \text{when } |k| = 3 \\ \dots & \dots \end{cases}\right) = \begin{cases} P_0 & \text{when } |k| = 0 \\ P_1 & \text{when } |k| = 1 \\ \lambda k.\ \bot & \text{when } |k| = 2 \\ \lambda k.\ \bot & \text{when } |k| = 3 \\ \dots & \dots \end{cases}$$

This is just "slicing" P into disjoint partial functions by partitioning its domain; the P_i are just how P behaves on knots of level i.

unsquash ∘ squash

So what is $\mathsf{unsquash} \circ \mathsf{squash}$? We approximate pointwise:

$$\mathsf{unsquash} \circ \mathsf{squash}(n, \phi) \quad = \quad (n, \mathsf{approx}_n \circ \phi)$$

That is,

$$\mathsf{unsquash} \circ \mathsf{squash} \qquad \left(n,\ \lambda\ell.\ \begin{cases} \tau_0 & \text{when } \ell = 0 \\ \tau_1 & \text{when } \ell = 1 \\ \tau_2 & \text{when } \ell = 2 \\ \tau_3 & \text{when } \ell = 3 \\ \dots & \dots \end{cases}\right)$$

$$= \qquad \left(n,\ \lambda\ell.\ \begin{cases} \mathsf{approx}_n(\tau_0) & \text{when } \ell = 0 \\ \mathsf{approx}_n(\tau_1) & \text{when } \ell = 1 \\ \mathsf{approx}_n(\tau_2) & \text{when } \ell = 2 \\ \mathsf{approx}_n(\tau_3) & \text{when } \ell = 3 \\ \dots & \dots \end{cases}\right)$$

Relating squash/unsquash to $\rightsquigarrow$/$|\cdot|$

The operations squash and unsquash are directly related to our "approximatable" operations $\rightsquigarrow$ and $|\cdot|$:

- The level of a knot is just the first projection of its unsquashing:

  $$|k| \ = \ \mathsf{fst}(\mathsf{unsquash}(k))$$

  The level gives the amount of "information" (circularity, recursion depth) in the knot.
- To go through the $\rightsquigarrow$ relation, unsquash and then resquash to the next lower level:

  $$k_1 \rightsquigarrow k_2 \ \leftrightarrow \ \begin{array}{l} \mathsf{let}\ (n+1, \phi) = \mathsf{unsquash}(k_1)\ \mathsf{in} \\ \quad k_2 = \mathsf{squash}(n, \phi) \end{array}$$

  This relation does not hold when $|k| = 0$. The effect is to "grind down" the types contained in the knot.
- This seems counterproductive. Why would we want to lose information?

Using a type pulled out of a knot

Suppose we have a knot k. We want to use it to discover what the type associated with address ℓ is; that is:

1. We unsquash k to get (n, ϕ)
2. We then lookup $\phi(\ell)$ to get the type τ associated with ℓ.

Notice that (since to create k in the first place we must have squashed some initial ϕ_i to n) we have that $\tau = \mathsf{approx}_n(\tau)$.

Now that we have τ, we want to use it—that is, to apply it to some world (k', v). The obvious knot to apply it to is k itself (that is, set $k' = k$). But here we have a problem: for all v, we have:

$$\tau(k, v) \;=\; \mathsf{approx}_{|k|}(\tau)(k, v) \;=\; \bot$$

That is, τ is **unable to judge the knot whence it came**.

Aging worlds; hereditariness

- The best we can do is apply τ to a more approximate knot—that is, if $k \rightsquigarrow k'$, then $\tau(k', v)$ can say something meaningful.
- Each time we use the knot, we must approximate it further. We handle arbitrarily-long execution traces by universal quantification on the initial level of k.
- We have a secondary problem.
  - Lift our $\rightsquigarrow$ and $|\cdot|$ operations from knots to worlds:

$$\begin{array}{rcl} |(k, v)| & \equiv & |k| \\ (k, v) \rightsquigarrow (k', v') & \equiv & k \rightsquigarrow k' \;\wedge\; v = v' \end{array}$$

  - Suppose we have $\tau(w)$ for some w, and we approximate w to w' (*i.e.*, $w \rightsquigarrow w'$). Need it be the case that $\tau(w')$?
  - Unfortunately the answer is no. This τ is not stable:

$$\tau \;\equiv\; \lambda w.\, |w| > 5$$

  - We say that a predicate that **does** have this property is *hereditary*; our soundness proofs contain numerous examples of proving that particular definitions (*e.g.*, for ref τ) are hereditary.

Semantic Types

We say τ_1 *entails* τ_2 if the truth of τ_1 forces the truth of τ_2 in all worlds.

$$\tau_1 \vdash \tau_2 \quad \equiv \quad \forall w : \mathbb{W}.\, w \models \tau_1 \to w \models \tau_2$$

Sometimes we need to compare two predicates (types) for equality. However, full equality is too strong because it fails to be hereditary. Instead we use a notion of *approximate equality*.

$$\tau_1 =_n \tau_2 \quad \equiv \quad \forall w.\, |w| < n \to (w \models \tau_1 \leftrightarrow w \models \tau_2)$$

Semantic Types (3)

Now we can already define some basic type constructors.

$(\phi, v) \models \texttt{nat} \equiv \exists n.\, v = \texttt{Nat}\ n$	*naturals*
$(\phi, v) \models \texttt{just}\ v' \equiv v = v'$	*singleton type*
$(\phi, v) \models \texttt{typeat}\ \ell\ \tau \equiv \phi(\ell) =_{\lvert\phi\rvert} \tau$	*address typing*
$\texttt{ref}\ \tau \equiv \exists \ell.\, \texttt{just}\ (\texttt{Loc}\ \ell) \wedge \texttt{typeat}\ \ell\ \tau$	*references*
$(\phi, v) \triangleright \tau \equiv \forall \phi'.\, \phi \rightsquigarrow^+ \phi' \to (\phi', v) \models \tau$	*approximately*
$(\phi, v) \% \tau \equiv \forall \phi'.\, \texttt{extends}\ \phi\ \phi' \to (\phi', v) \models \tau$	*extendedly*
$(\phi, v) \diamond \tau \equiv \exists \phi'.\, \texttt{extends}\ \phi\ \phi' \wedge (\phi', v) \models \tau$	*dual of* $\%$

$\triangleright$, $\%$ and $\diamond$ are important *modalities* which alter the meaning of a type.

- $\triangleright\tau$ means τ holds in all *strictly* more approximate worlds
- $\%\tau$ means τ holds in *all* worlds where additional reference types have been added
- $\diamond\tau$ means that τ holds in *some* world with extended reference types

Gödel-Löb Rule

Because of the way we set up our approximation structure on worlds $\triangleright$ behaves in a very special way. It enjoys an induction rule called the Gödel-Löb rule.

$$\frac{P \wedge \triangleright Q \vdash Q}{P \vdash Q}\ \textit{Gödel-Löb}$$

The $\triangleright$ operator weakens a predicate in just the right way so that it is appropriate as an induction hypothesis.

This induction rule is one of the key pieces of our final safety theorem. It is what allows the whole approximation approach to hang together.

Typing Expressions (1)

Notice that we defined types as hereditary predicates on worlds, which contain values. However, we want to type *expressions*, not just values. This turns out to be necessary to define the function type as well.

Roughly, we want to say that an expression e has type τ if it evaluates to a value having type τ.

First, we need to say when a memory satisfies a memory typing ϕ. Roughly, every allocated reference must satisfy (approximately and in all extended worlds) the type stored in ϕ.

$$(\phi, v) \models \texttt{validmem}\, m \ \equiv\ \forall \ell.\, (\phi, m(\ell)) \models \% \triangleright (\phi(\ell))$$

Typing Expressions (2)

Expression typing is captured by the following recursive definition.

$$\begin{aligned}&\texttt{expr\_type}\ e\ \tau \ \equiv\ \%\forall m.\ \texttt{validmem}\ m \Rightarrow \\ &\quad (\forall m'\ e'.\ (m,e) \longrightarrow (m',e') \Rightarrow \triangleright \diamond (\texttt{validmem}\ m' \wedge \texttt{expr\_type}\ e'\ \tau)) \\ &\quad \wedge \\ &\quad ((m,e)\Downarrow \Rightarrow \texttt{isValue}\ e \wedge \texttt{withval}\ e\ (\%\tau))\end{aligned}$$

The definition breaks down into two mutually-exclusive cases.

1. The expression can take a step. In this case, we say that the new memory m' is valid *in some potentially extended memory type* and expression e' recursively has type τ.
2. The expression cannot step. Then the expression must be a value of the appropriate type.

In case 1, allowing the memory type to extend allows for the case where evaluating e caused an allocation.

The Function Type

Now, with a definition for expression typing in hand, we can define the critical arrow type constructor.

$$\begin{aligned}&\texttt{lam}\ \tau_1\ \tau_2 \ \equiv\ \exists e.\ \texttt{just}\ (\texttt{Lam}\ e) \wedge \\ &\quad \triangleright\%(\forall v.\ \texttt{withval}\ v\ (\%\tau_1) \Rightarrow (\texttt{expr\_type}\ (\texttt{subst}\ v\ e)\tau_2))\end{aligned}$$

The first part of the definition simply asserts that the value must be a λ abstraction. The second part is the meat: it says that (under approximation) whenever a value of type τ_1 is substituted into e, the resulting expression has type τ_2.

Semantic Typing Judgment

It is a simple matter to lift value typing to *value environments*, which are just lists of values. A value environment is typed by a type environment (a list of types).

$$(\phi, \_) \models \vec{v} : \Gamma \quad \equiv \quad |\vec{v}| = |\Gamma| \wedge \forall n.(\phi, v_n) \models \Gamma_n$$

Here $\vec{v}$ is a list of values and Γ is a list of types. $\vec{v} : \Gamma$ holds when the lists have the same length and corresponding elements stand in the typing relation.

Now we can define the semantic typing judgment.

$$\Gamma \vdash e : \tau \quad \equiv \quad \mathtt{fv}\, e \leq |\Gamma| \wedge (\forall \vec{v}.\, (\vec{v} : \Gamma) \vdash \mathtt{expr\_type}\, (\mathtt{subst}\, \vec{v}\, e)\, \tau)$$

Thus e has type τ under Γ iff for all closing value environments $\vec{v}$ of type Γ, $\mathtt{subst}\, \vec{v}\, e$ has type τ. (NB: the $\vdash$ on the RHS is predicate entailment).

The safety theorem

Theorem (Program Safety)

For all e and τ such that $\cdot \vdash e : \tau$, e is a safe program.

Note that the **result** of the central theorem is the the safety policy from the TCB.

The **premise** of our safety theorem is that an expression e has type τ in an empty typing context, written $\cdot \vdash e : \tau$.

The safety theorem follows directly from the definition of the typing judgment and induction on the level of approximation.

Our typing rules

How do we prove that a given expression e is well typed in some context Γ?

In the usual way–by using typing rules. Actually, all of our rules are exactly standard. The difference is that we prove the rules as **lemmas** from our semantic definitions.

The end result is end-to-end: from a program e, we use typing rules to give it a type τ. Our typing rules are proved sound—thus, we know that if our typing rules claim that $\cdot \vdash e : \tau$, it is actually the case (semantically) that e is well-typed.

Finally, our safety theorem tells us that anything that is (semantically) well-typed meets our safety policy.

Outline

Example 2: Main take-away ideas

- Separation algebras as semantics for separation logic
- The continuation-based hoare tuple
- Functionwise whole-program verification

The one-slide intro to separation logic

Separation logic (SL): a Floyd-Hoare logic where the assertion language is (some extension of) the logic of Bunched Implications.

$$P * Q$$

P holds and Q holds and furthermore they reference *disjoint* resources.

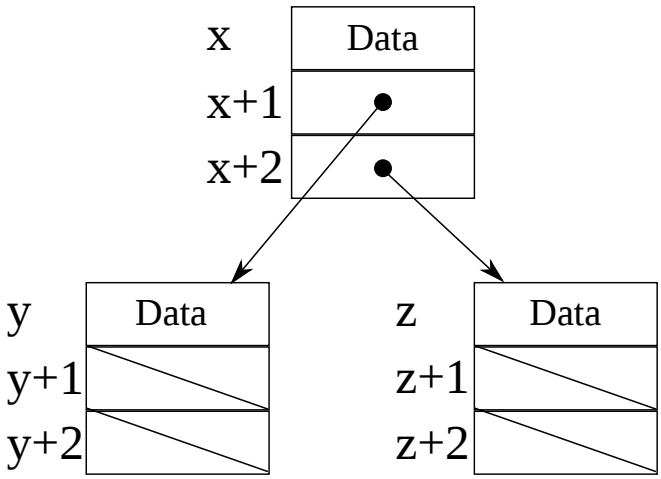
The “points-to” operator describes the contents of a cell in memory:

$$\ell \mapsto v$$

Inductive data in SL

SL is good at describing the shape of tree-structured data:

$$\texttt{tree}(\ell) \equiv \ell = \texttt{null} \vee (\ell \mapsto \_ * (\ell+1) \mapsto \ell_1 * (\ell+2) \mapsto \ell_2 * \texttt{tree}(\ell_1) * \texttt{tree}(\ell_2))$$



Separation Algebras

Separation algebras (SAs) are structures that capture the notion of separable resources.
A SA is consists of a carrier set A and a partial operation $\oplus$ which:

1. is commutative $x \oplus y = y \oplus x$
2. is associative $x \oplus (y \oplus z) = (x \oplus y) \oplus z$
3. is cancellative $x_1 \oplus y = x_2 \oplus y \rightarrow x_1 = x_2$
4. has units $\forall x.\ \exists x_u.\ x_u \oplus x = x$
5. is self-disjoint $x \oplus x = y \rightarrow x = y$

Whenever $x \oplus y = z$ we say that x and y are *disjoint* and that x and y *join* to make z.

SL from SA

Given a SA (and an approximation structure) on A, we can define the operators of separation logic.

$$
\begin{array}{lcl}
w \models \texttt{emp} & \equiv & w \oplus w = w \\
w \models p * q & \equiv & \exists w_1\ w_2,\ (w_1 \oplus w_2 = w) \wedge \\
 & & \quad (w_1 \models p) \wedge (w_2 \models q) \\
w \models p \mathrel{-\!\!*} q & \equiv & \forall w_1\ w_2\ w_3,\ (w \rightsquigarrow^* w_1) \to (w_1 \oplus w_2 = w_3) \to \\
 & & \quad (w_2 \models p) \to (w_3 \models q)
\end{array}
$$

Nutshell: Defining a separation algebra gives us an automatic way to define a well-behaved separation logic.

While programs, basics

$$
\begin{array}{llll}
i & ::= & \mathbb{N} & \textit{identifiers} \\
a & ::= & \mathbb{N} & \textit{addresses} \\
v & ::= & i + a & \textit{values} \\
\pi & ::= & \cdots & \textit{shares} \\
 & & & \\
\rho & ::= & i \rightharpoonup v & \textit{local env} \\
 & & & \\
m & ::= & a \rightharpoonup (\pi \times v) & \textit{memory}
\end{array}
$$

While programs, syntax

$$
\begin{array}{lcll}
e & ::= & \{f : \rho \rightharpoonup v \mid f \text{ is monotone}\} & \textit{expressions} \\
\\
c & ::= & \texttt{skip} & \textit{commands} \\
 & \mid & c_1 \,;\, c_2 & \\
 & \mid & i \;:=\; f(\vec{e}) & \\
 & \mid & \texttt{ret}\, e & \\
 & \mid & i \;:=\; e & \\
 & \mid & i \;:=\; [e] & \\
 & \mid & [e_1] \;:=\; e_2 & \\
 & \mid & \texttt{if}\, e\, \texttt{then}\, c_1\, \texttt{else}\, c_2 & \\
 & \mid & \texttt{while}\, e\, \texttt{do}\, c & \\
\\
fd & ::= & \vec{i} \times \overrightarrow{(i \times v)} \times c & \textit{function declarations} \\
 & & (\text{formals},\ \text{local decls},\ \text{command}) & \\
p & ::= & i \rightharpoonup fd & \textit{programs}
\end{array}
$$

While programs, operational semantics summary

$$\boxed{(\kappa, \rho, m) \xrightarrow{p} (\kappa', \rho', m')}$$

$$
\begin{array}{lcl}
\kappa & ::= & \texttt{kseq}\, c\, \kappa \\
 & \mid & \texttt{kcall}\, i\, \rho\, \kappa \\
 & \mid & \texttt{knil}
\end{array}
$$

$$\frac{e(\rho) = \texttt{int}\, x \qquad x \neq 0}{(\texttt{kseq}\,(\texttt{if}\, e\, \texttt{then}\, c_1\, \texttt{else}\, c_2)\, \kappa, \rho, m) \xrightarrow{p} (\texttt{kseq}\, c_1\, \kappa, \rho, m)}\ \textit{IfTrue}$$

$$\frac{e(\rho) = \texttt{adr}\, a \qquad m(a) = (\pi, v)}{(\texttt{kseq}\,(i \;:=\; [e])\, \kappa, \rho, m) \xrightarrow{p} (\kappa, \rho[i \leftarrow v], m)}\ \textit{Load}$$

$$\frac{p(f) = (frms, dcls, c) \qquad \vec{e}(\rho) = \vec{v}}{\begin{array}{l}(\texttt{kseq}\,(i \;:=\; f(\vec{e}))\, \kappa, \rho, m) \xrightarrow{p} \\ \quad (\texttt{kseq}\, c\, (\texttt{kcall}\, i\, \rho\, \kappa), \texttt{locals}(frms, \vec{v}, dcls), m)\end{array}}\ \textit{Call}$$

Worlds for the imperative language

Like with the λ -calculus, we define an assertion language as predicates on "worlds."

$$\mathbb{W} \equiv \mathbb{N} \times \rho \times m$$

A world is a tuple of a natural number, a local variable environment and memory.

The nat allows us to define the notion of approximation we need to get the Gödel-Löb operator, $\triangleright$.

SA on memories

Two memories join if cells at every address join. Cells join if their shares add up and they have the same value. Empty cells join with any cell. Think of shares as numbers between 0 and 1.

0	(0.5, `int` 10)
1	(1, `adr` 24)
2	
3	(0.5, `int` 7)

$\oplus$

0	(0.5, `int` 10)
1	
2	
3	(0.25, `int` 7)

=

0	(1, `int` 10)
1	(1, `adr` 24)
2	
3	(0.75, `int` 7)

A nonzero share allows read access. Full share allows exclusive write access.

SA on local variables

For locals we have two basic choices:

1. Treat local variables as resources, using separation logic to handle freshness.
2. Treat locals as nonresources, using side conditions to handle freshness.

Option 2 is how traditional hoare logics handle freshness. It is a pretty "syntactic" method.

We will take option 1 (variables-as-resources) for the sake of novelty and to explore the possibilities.

Thus, the SA on locals is basically like the SA on memories, but without shares.

Basic formulae

$a \overset{\pi}{\mapsto} v$	Address a has value v, share π
$i \mapsto v$	local i has value v
$i \mapsto \_$	local i has some value
$e \Downarrow v$	expression e evaluates to v
`anylocals`	Accepts any ρ, requires empty m
`lift` P	P holds in all ρ and m

$a \overset{\pi}{\mapsto} v$ and $i \mapsto v$ are *tight* specifications, indicating that all other cells/locals are empty.

$e \Downarrow v$ is *not* tight and holds in environments larger than required to evaluate e.

The `lift` predicate takes a predicate on worlds $\mathbb{W}$ and turns it into a predicate on $\mathbb{N}$.

$$n \models \texttt{lift}\, P \quad \equiv \quad \forall \rho\, m.\, (n, \rho, m) \models P$$

The semantic hoare triple (1)

We reduce the hoare tuple of *partial* correctness into the more primitive notion of program safety.

A program state (κ, ρ, m) is *safe* in program p for n steps provided: for all $m \leq n$ and tuples (κ', ρ', m'), where (κ, ρ, m) steps to (κ', ρ', m') in exactly m steps **either** $\kappa' = \texttt{knil}$ **or** (κ', ρ', m') can take another step.

$$(n, \rho, m) \models \texttt{safen}\, p\, \kappa$$

The `safen` predicate expresses that the state (κ, ρ, m) is safe in p for n steps.

The semantic hoare triple (2)

A predicate *guards* a continuation κ if the predicate makes κ safe to run. That is, P is a precondition for executing κ.

$$\texttt{guards}\, p\, P\, \kappa \;\equiv\; \texttt{lift}\, (P \Rightarrow \texttt{safen}\, p\, \kappa)$$

We'll also need a more technical definition for "return guards:"

$$\texttt{rguards}\, p\, R\, F\, \kappa \;\equiv\; \forall e.\, \texttt{guards}\, p\, ((\exists v.\, e \Downarrow v \wedge R\, v) * F)\, \kappa$$

Here $R : \mathit{val} \to \texttt{pred}\, \mathbb{W}$ and $F : \texttt{pred}\, \mathbb{W}$. Basically, R represents the postcondition that must be true on function return to control stack κ.

The semantic hoare triple (3)

$$\begin{aligned}&\texttt{hoare}\ p\ R\ P\ c\ Q\ \equiv\\ &\quad\forall\kappa\ F.\\ &\qquad\texttt{rguards}\ p\ R\ F\ \kappa \Rightarrow\\ &\qquad\texttt{guards}\ p\ (Q * F)\ \kappa \Rightarrow\\ &\qquad\texttt{guards}\ p\ (P * F)\ (\texttt{kseq}\ c\ \kappa)\end{aligned}$$

Here $R : \mathit{val} \to \texttt{pred}\ \mathbb{W}$ and $P, Q : \texttt{pred}\ \mathbb{W}$.

This continuation-passing definition simplified is “whenever the postconditions make the continuation κ safe, the preconditions make running c before entering κ safe.

Note, the first order frame rule is baked directly into the definition.

Hoare Rules, summary

$$\frac{\texttt{hoare}\ p\ R\ P\ c\ Q}{\texttt{hoare}\ p\ (\lambda v.\ R\ v * F)\ (P * F)\ c\ (Q * F)}\ \mathit{HFrame}$$

$$\frac{}{\begin{array}{l}\texttt{hoare}\ p\ R\\ \quad(\exists a\ \pi\ v.\ (e \Downarrow \texttt{adr}\ a) \wedge\\ \qquad(a \overset{\pi}{\mapsto} v) * (i \mapsto \_) * (a \overset{\pi}{\mapsto} v * i \mapsto v \mathrel{-\!\!*} Q))\\ \quad(i\ :=\ [e])\\ \quad Q\end{array}}\ \mathit{Hload}$$

$$\frac{}{\texttt{hoare}\ p\ R\ (\exists v.\ (e \Downarrow v) \wedge R\ v)\ (\texttt{ret}\ e)\ Q}\ \mathit{Hret}$$

$$\frac{\texttt{satisfies\_fun\_spec}\ p\ f\ \mathit{fs}}{\begin{array}{l}\texttt{hoare}\ p\ R\\ \quad(\exists \vec{v}\ a.\ (\vec{e} \Downarrow \vec{v}) \wedge \texttt{pre}_{\mathit{fs}}\ a\ \vec{v} * (i \mapsto \_) *\\ \qquad(\forall v.\ \texttt{post}_{\mathit{fs}}\ a\ v * (i \mapsto v) \mathrel{-\!\!*} Q))\\ \quad(i\ :=\ f(\vec{e}))\\ \quad Q\end{array}}\ \mathit{Hcall}$$

Program Verification (1)

We verify functions against their *specifications*, which specify the function pre- and post-conditions.

For each function f in the program we choose a *function specification* fs with components: type A_{fs}, a pre-condition $\texttt{pre}_{fs} : A_{fs} \to \vec{v} \to \texttt{pred}\ \mathbb{W}$ and a post-condition $\texttt{post}_{fs} : A_{fs} \to v \to \texttt{pred}\ \mathbb{W}$.

The type A_{fs} captures what information is shared between the pre and post. The verifier at the call site chooses an appropriate A_{fs}.

Technical restriction: the pre and post require an empty local variable environment.

Program Verification (2)

Main Idea: to verify a program, first **assume** that every function *approximately* satisfies its specification, and then **show** that every function *actually* satisfies its specification.

Intuition: approximate facts hold after we take at least one more step of computation. Calling a function consumes a step, so the approximate assumption suffices to reason about call sites.

The approximate assumption allows us to verify function call sites without begging the question. The Gödel-Löb rule ties the knot by induction on the approximation index.

Program Verification (3)

$$ps ::= f \rightharpoonup fs \qquad \textit{Program Specifications}$$

$$\begin{aligned}
&\texttt{satisfies_fun_spec}\ p\ f\ fs \equiv \\
&\quad \exists fd.\ p(f) = fd\ \wedge \\
&\qquad \forall a : A_{fs}.\ \triangleright\, \texttt{hoare}\ p \quad (\lambda v.\ \texttt{post}_{fs}\ a\ v * \texttt{anylocals}) \\
&\qquad\qquad\qquad\qquad\ (\exists \vec{v}.\ \texttt{pre}_{fs}\ a\ \vec{v} * \texttt{func_locals}\ fd\ \vec{v}) \\
&\qquad\qquad\qquad\qquad\ c_{fd} \\
&\qquad\qquad\qquad\qquad\ \bot
\end{aligned}$$

$$\begin{aligned}
&\texttt{satisfies_spec}\ p\ ps \equiv \\
&\quad \forall f\ fs.\ (ps(f) = fs) \Rightarrow \texttt{satisfies_fun_spec}\ p\ f\ fs
\end{aligned}$$

Program Verification (4)

$$\begin{aligned}
&\texttt{validate}\ ps\ p\ ps' \equiv \\
&\quad \triangleright \texttt{satisfies_spec}\ p\ ps \vdash \texttt{satisfies_spec}\ p\ ps'
\end{aligned}$$

The `validate` predicate captures a state of partial verification. We have assumed all of ps but only proved ps', which will typically be a subset of ps.

When verification is complete, $ps = ps'$. The Gödel-Löb rule then shows that the program satisfies its spec.

Theorem (Program Safety)

Whenever `validate` *ps p ps holds, It is safe to call any function mentioned in ps in any environment satisfying the stated precondition.*

Trivial base case to get verification started:

$$\frac{}{\texttt{validate}\ ps\ p\ (\lambda f.\ \bot)}\ \textit{VEmpty}$$

Interesting case: verify a single function body, add it to the set of verified functions.

$$\frac{\begin{array}{l} p(i) = fd \\ \texttt{validate}\ ps\ p\ ps' \\ \left(\begin{array}{l} \texttt{satisfies\_spec}\ p\ ps \vdash \forall a : A_{fs}. \\ \quad \texttt{hoare}\ p \quad (\lambda v.\ \texttt{post}_{fs}\ a\ v * \texttt{anylocals}) \\ \qquad\qquad\ \ (\exists \vec{v}.\ \texttt{pre}_{fs}\ a\ \vec{v} * \texttt{func\_locals}\ fd\ \vec{v}) \\ \qquad\qquad\ \ c_{fd} \\ \qquad\qquad\ \ \bot \end{array}\right) \end{array}}{\texttt{validate}\ ps\ p\ (ps' \cdot f \mapsto fs)}\ \textit{VSingle}$$

Outline

Cminor is a C-like language that forms one of the high-level stages of the CompCert verified compiler. It supports a quite large subset of the functionality of C.

Concurrent Cminor is an extension of Cminor with support for threads-and-locks style concurrency.

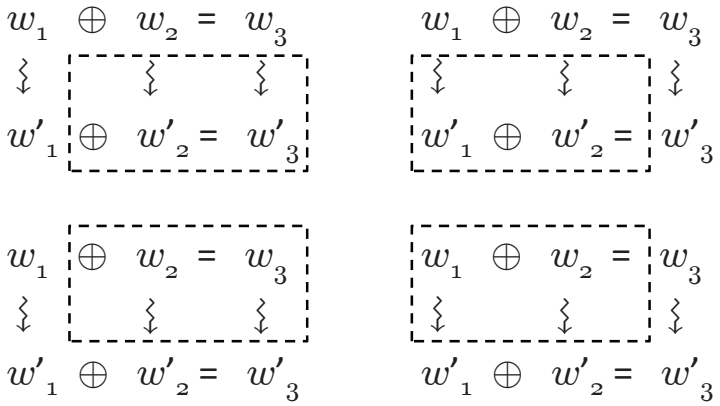
Proving the soundness of the program logic for this language is rather complicated and forms the original motivation for most of the work in this tutorial.

For our purposes today, CCm is interesting because it requires both predicates-in-the-heap and separation logic.

Mixing separation and approximation (1)

To get separation and approximation in the same logic, we need an approximation structure and a separation algebra on the same set of worlds.

We also need to restrict their interaction so they “play nice” together. We need the join relation and the age relation to commute. The 4 diagrams below show the axioms we need; the elements in the dotted boxes are asserted to exist.



Mixing separation and approximation (2)

Fortunately, the required axioms are not too difficult to prove.

In return, they are used to show the definitions of the separation logic operators from the last section are hereditary, and they also are used to prove nice equations involving both approximation and separation.

$$\begin{aligned} \triangleright(P * Q) &= \triangleright P * \triangleright Q \\ \triangleright(P \mathrel{-\!\!*} Q) &= \triangleright P \mathrel{-\!\!*} \triangleright Q \end{aligned}$$

Locks and Invariants

Locking protocols are handled in the logic of CCm using a resource transfer analogy. A lock controls some resources (e.g., a shared data structure). When a thread acquires a lock, it obtains the controlled resources. It relinquishes the resources when releasing the lock.

Acquire lock = get resources
Release lock = relinquish resources

Each lock has a *resource invariant* which describes the controlled resources.

Lock rules (simplified)

$$\frac{}{\texttt{hoare}\ (\ell \overset{\pi}{\rightsquigarrow} I)\ (\texttt{lock}\ \ell)\ (\ell \overset{\pi}{\rightsquigarrow} I * I * \texttt{hold}\ \ell)}\ \textit{Lock}$$

$$\frac{}{\texttt{hoare}\ (\ell \overset{\pi}{\rightsquigarrow} I * I * \texttt{hold}\ \ell)\ (\texttt{unlock}\ \ell)\ (\ell \overset{\pi}{\rightsquigarrow} I)}\ \textit{Unlock}$$

$$\frac{\texttt{precice}\ I}{\texttt{hoare}\ (\ell \overset{1}{\mapsto} 0)\ (\texttt{makelock}\ \ell\ I)\ (\ell \overset{1}{\rightsquigarrow} I * \texttt{hold}\ \ell)}\ \textit{Makelock}$$

The special `hold` resource represents the ability to unlock a lock. This ensures lock acquire/release is well-bracketed.

Precise predicates are those which identify a unique subset of a world. This technical restriction is necessary to make the soundness result work out.

Predicates in the heap, again

To give semantics to the lock predicate $\ell \rightsquigarrow I$, we need to store the invariant I in the memory.

I is an arbitrary (precise) formula in separation logic, so we need to store predicates in the memory, which get judged by predicates...

We again use indirection theory to approximate the desired domain for building our worlds.

Too many details!

Unfortunately, CCm is much too large to go into much detail in a talk of this length. Some salient points:

- Memory is both an approximation and a separable structure.
- The hoare tuple is defined in a similar continuation-passing style as before.
- The logic of CCm enforces a data-race-free discipline, but can still allow shared reads using share accounting.
- Considerably more complicated, but the same techniques from the simple examples still apply!

See Aquinas Hobor's thesis for all the gritty details.

Summary

What we hope you take away:

- Semantic methods for PL theory have a different flavor than syntactic/subject-reduction and is a useful tool to consider.
- Approximation is a useful tool for dealing with semantic modeling problems where one wishes to associate invariants with addressable storage.
- Separation logic is a powerful reasoning tool for languages with addressable storage.
- The MSL is a Coq proof library which can help you build machine-verified proofs for program logics.

Outline

While programs, operational semantics (1)

$$\boxed{(\kappa, \rho, m) \xrightarrow{p} (\kappa', \rho', m')}$$

$$\begin{array}{rcl} \kappa & ::= & \texttt{kseq}\, c\, \kappa \\ & | & \texttt{kcall}\, i\, \rho\, \kappa \\ & | & \texttt{knil} \end{array}$$

$$\frac{}{(\texttt{kseq}\,(\texttt{skip})\,\kappa, \rho, m) \xrightarrow{p} (\kappa, \rho, m)}\ \textit{Skip}$$

$$\frac{}{(\texttt{kseq}\,(c_1\ ;\ c_2)\,\kappa, \rho, m) \xrightarrow{p} (\texttt{kseq}\, c_1\, (\texttt{kseq}\, c_2\, \kappa), \rho, m)}\ \textit{Seq}$$

$$\frac{e(\rho) = \texttt{int}\, x \qquad x \neq 0}{(\texttt{kseq}\,(\texttt{if}\, e\, \texttt{then}\, c_1\, \texttt{else}\, c_2)\,\kappa, \rho, m) \xrightarrow{p} (\texttt{kseq}\, c_1\, \kappa, \rho, m)}\ \textit{IfTrue}$$

$$\frac{e(\rho) = \texttt{int}\, 0}{(\texttt{kseq}\,(\texttt{if}\, e\, \texttt{then}\, c_1\, \texttt{else}\, c_2)\,\kappa, \rho, m) \xrightarrow{p} (\texttt{kseq}\, c_2\, \kappa, \rho, m)}\ \textit{IfFalse}$$

While programs, operational semantics (2)

$$\frac{e(\rho) = v \qquad \rho(i) \text{ defined}}{(\texttt{kseq}\ (i\ :=\ e)\ \kappa, \rho, m) \xrightarrow{p} (\kappa, \rho[i \leftarrow v], m)}\ \textit{Assign}$$

$$\frac{e(\rho) = \texttt{adr}\ a \qquad m(a) = (\pi, v)}{(\texttt{kseq}\ (i\ :=\ [e])\ \kappa, \rho, m) \xrightarrow{p} (\kappa, \rho[i \leftarrow v], m)}\ \textit{Load}$$

$$\frac{e_1(\rho) = \texttt{adr}\ a \qquad e_2(\rho) = v \qquad m(a) = (1, \_)}{(\texttt{kseq}\ ([e_1]\ :=\ e_2)\ \kappa, \rho, m) \xrightarrow{p} (\kappa, \rho, m[a \leftarrow (1, v)])}\ \textit{Store}$$

$$\frac{e(\rho) = \texttt{int}\ x \qquad x \neq 0}{\begin{array}{l}(\texttt{kseq}\ (\texttt{while}\ e\ \texttt{do}\ c)\ \kappa, \rho, m) \xrightarrow{p} \\ \quad (\texttt{kseq}\ c\ (\texttt{kseq}\ (\texttt{while}\ e\ \texttt{do}\ c)\ \kappa), \rho, m)\end{array}}\ \textit{WhileTrue}$$

$$\frac{e(\rho) = \texttt{int}\ 0}{(\texttt{kseq}\ (\texttt{while}\ e\ \texttt{do}\ c)\ \kappa, \rho, m) \xrightarrow{p} (\kappa, \rho, m)}\ \textit{WhileFalse}$$

While programs, operational semantics (3)

$$\frac{p(f) = (\mathit{frms}, \mathit{dcls}, c) \qquad \vec{e}(\rho) = \vec{v}}{\begin{array}{l}(\texttt{kseq}\ (i\ :=\ f(\vec{e}))\ \kappa, \rho, m) \xrightarrow{p} \\ \quad (\texttt{kseq}\ c\ (\texttt{kcall}\ i\ \rho\ \kappa), \texttt{locals}(\mathit{frms}, \vec{v}, \mathit{dcls}), m)\end{array}}\ \textit{Call}$$

$$\frac{e(\rho) = v \qquad \texttt{unwind}(\kappa) = (i, \rho', \kappa')}{(\texttt{kseq}\ (\texttt{ret}\ e)\ \kappa, \rho, m) \xrightarrow{p} (\kappa', \rho'[i \leftarrow v], m)}\ \textit{Ret}$$

$$\begin{array}{lcl}\texttt{unwind}(\texttt{kseq}\ c\ \kappa) & = & \texttt{unwind}(\kappa) \\ \texttt{unwind}(\texttt{kcall}\ i\ \rho\ \kappa) & = & (i, \rho, \kappa)\end{array}$$

Hoare Rules (1)

$$\frac{\texttt{hoare}\ p\ R\ P\ c\ Q}{\texttt{hoare}\ p\ (\lambda v.\ R\ v * F)\ (P * F)\ c\ (Q * F)}\ \textit{HFrame}$$

$$\frac{P' \Rightarrow P \quad Q \Rightarrow Q' \quad (\forall v.\ R\ v \Rightarrow R'\ v) \qquad \texttt{hoare}\ p\ R\ P\ c\ Q}{\texttt{hoare}\ p\ R'\ P'\ c\ Q'}\ \textit{HConsequence}$$

$$\frac{\texttt{hoare}\ p\ R\ P_1\ c_1\ P_2 \qquad \texttt{hoare}\ p\ R\ P_2\ c_2\ P_3}{\texttt{hoare}\ p\ R\ P_1\ (c_1; c_2)\ P_3}\ \textit{Hseq}$$

$$\frac{}{\texttt{hoare}\ p\ R\ Q\ \texttt{skip}\ Q}\ \textit{Hskip}$$

Hoare Rules (2)

$$\frac{}{\begin{array}{l}\texttt{hoare}\ p\ R \\ \quad (\exists v.\ e \Downarrow v \wedge (i \mapsto \_) * (i \mapsto v \mathrel{-\!\!*} Q)) \\ \quad (i\ :=\ e) \\ \quad Q\end{array}}\ \textit{Hassign}$$

$$\frac{}{\begin{array}{l}\texttt{hoare}\ p\ R \\ \quad (\exists a\ v.\ (e_1 \Downarrow \texttt{adr}\ a) \wedge (e_2 \Downarrow v) \wedge (a \overset{1}{\mapsto} \_) * (a \overset{1}{\mapsto} v \mathrel{-\!\!*} Q)) \\ \quad ([e_1]\ :=\ e_2) \\ \quad Q\end{array}}\ \textit{Hstore}$$

$$\frac{}{\begin{array}{l}\texttt{hoare}\ p\ R \\ \quad (\exists a\ \pi\ v.\ (e \Downarrow \texttt{adr}\ a) \wedge \\ \qquad (a \overset{\pi}{\mapsto} v) * (i \mapsto \_) * (a \overset{\pi}{\mapsto} v * i \mapsto v \mathrel{-\!\!*} Q)) \\ \quad (i\ :=\ [e]) \\ \quad Q\end{array}}\ \textit{Hload}$$

Hoare Rules (3)

$$\frac{\texttt{hoare}\ p\ R\ (\exists n.\ (e \Downarrow \texttt{int}\ n) \wedge n \neq 0 \wedge P)\ c_1\ Q \qquad \texttt{hoare}\ p\ R\ (e \Downarrow \texttt{int}\ 0 \wedge P)\ c_2\ Q}{\texttt{hoare}\ p\ R\ (\exists n.\ (e \Downarrow \texttt{int}\ n) \wedge P)\ (\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2)\ Q}\ \mathit{Hif}$$

$$\frac{\texttt{hoare}\ p\ R\ (\exists n.\ (e \Downarrow \texttt{int}\ n) \wedge n \neq 0 \wedge I)\ c\ (\exists n.\ (e \Downarrow \texttt{int}\ n) \wedge I)}{\texttt{hoare}\ p\ R\ (\exists n.\ (e \Downarrow \texttt{int}\ n) \wedge I)\ (\texttt{while}\ e\ \texttt{do}\ c)\ (e \Downarrow \texttt{int}\ 0 \wedge I)}\ \mathit{Hwhile}$$

Hoare Rules (4)

$$\frac{}{\texttt{hoare}\ p\ R\ (\exists v.\ (e \Downarrow v) \wedge R\ v)\ (\texttt{ret}\ e)\ Q}\ \mathit{Hret}$$

$$\frac{\texttt{satisfies\_fun\_spec}\ p\ f\ fs}{\begin{array}{l}\texttt{hoare}\ p\ R \\ \quad (\exists \vec{v}\ a.\ (\vec{e} \Downarrow \vec{v}) \wedge \texttt{pre}_{fs}\ a\ \vec{v} * (i \mapsto \_) * \\ \qquad (\forall v.\ \texttt{post}_{fs}\ a\ v * (i \mapsto v) \mathrel{-\!\!*} Q)) \\ \quad (i\ :=\ f(\vec{e})) \\ \quad Q\end{array}}\ \mathit{Hcall}$$